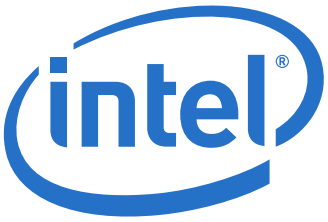


Container and Kernel-Based Virtual Machine (KVM) Virtualization for Network Function Virtualization (NFV)

White Paper

August 2015

Order Number: 332860-001US

Contents

Figures

Tables

Revision History

Date	Revision	Description
August 2015	001	Initial release

1.0 Introduction

Recently, Linux* containers have gained significant attention from the community with projects like Docker (Docker, n.d.) growing significantly in feature set and user base. Since the introduction of Docker, the developer community has embraced containers as an alternative to hypervisor-based virtualization for certain workloads. Linux containers introduce less overhead, isolating applications from each other, instead of virtualizing the hardware. These benefits come from the simple fact that each container interacts with the same kernel as the host system instead of with a different kernel that is running on top of a hypervisor. The lower amount of overhead introduced by Linux containers allows running more containers on a system compared to VMs running with hypervisors.

Choosing the right virtualization technology is key for realizing the potential of virtualization. This white paper provides an overview of container technology, its current pain points and its strengths compared to hypervisor-based virtualization.

This paper will investigate if the reduced overhead of containers materializes into actual performance gain, and what the cost of running containers are. A concern paired with a container is that of security as the attack surface of containers are larger. The discussion will also focus on mitigation techniques limiting this threat.

1.1 Key Objectives

- Provides an introduction to the fundamental concepts of Linux containers.
- Highlights some of the platform-level considerations and comparison between containers and virtual machines.
- Details the usage of the Intel® Data Plane Development Kit (DPDK, n.d.) inside containers.
- Documents throughput and latency benchmarks of certain workloads.
- Discusses the challenge brought forward by the lack of a hypervisor.

1.2 References

Table 1. References

Title	Location
Verma, et al. (2015). Large-scale cluster management at Google with Borg.	http://static.googleusercontent.com/media/research.google.com/en/us/pubs/archive/43438.pdf
Narayanan. (2014). Tupperware: containerized deployment at Facebook.	http://www.slideshare.net/dotCloud/ tupperware-containerized-deployment-at-facebook
Intel. Linux* Containers Streamline Virtualization and Complement Hypervisor-Based Virtual Machines.	http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/linux-containers-hypervisor-based-vms-paper.pdf

2.0 An Introduction to Containers

Container-based virtualization refers to a set of processes running independently of or isolated from each other on a single Linux system. This is done using the Linux kernel control group (CGroup) and namespace mechanisms. This shared kernel approach is in contrast to hypervisor-based virtualization where each guest has its own kernel.

Figure 1. Representation of Two Containers

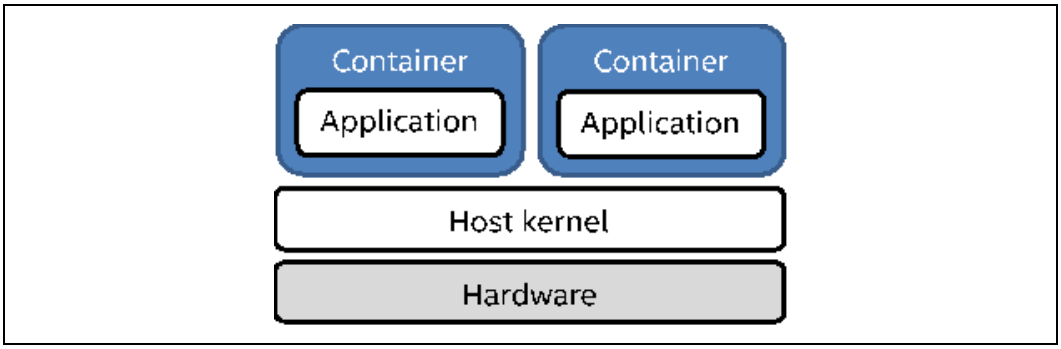
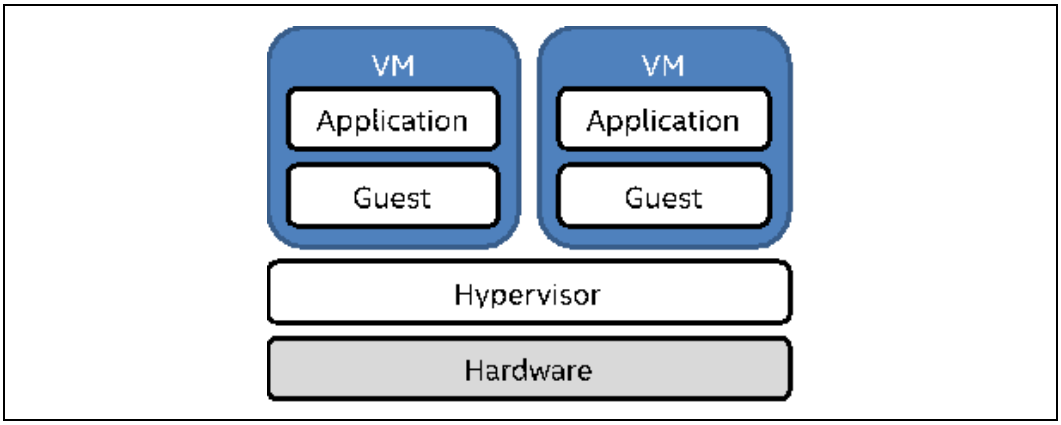


Figure 2. Representation of Two Virtual Machines



2.1 Namespaces

Namespaces are used to divide the processes on a system into groups of processes that can only see resources of other processes in the same group. There is a namespace for each of the subsystems in the Linux kernel that have support for it. Currently, there are six types of namespaces in the kernel and each manages the control over a set of global resources.

2.1.1 Mount Namespace

The mount namespaces isolates mount points seen by the different processes. Each namespace is represented with their own set of mount points. This is what allows setting up a container with full access to its own root partition, just like when using a virtual machine.

2.1.2 Net Namespace

The network namespaces operate on the networking interfaces available to a system. It allows assigning network interfaces to a specific namespace so that only the processes in that namespace will be able to configure and use them. Other processes on the system will not even see that the devices assigned to a different namespace are present.

2.1.3 UTS Namespace

UNIX* Time-Sharing (UTS) namespaces allow setting a different host name per namespace. This namespace primarily operates on the result returned from the Linux uname() system call.

2.1.4 IPC Namespace

The IPC namespaces ensure that the identifiers used during inter-process communication are unique for each namespace in which they are used. This means that there is no way for processes in one namespace to start communicating with processes in another using System-V IPC or POSIX message queues.

2.1.5 PID Namespace

The PID namespaces allow setting up a mapping between process IDs in one namespace to and its parent namespace. The mapping allows the ID of a process to start from 1 and thus allowing processes such as INIT to run inside a container. Inside the namespace all the processes see the internal process ID, which is the mapped ID. Processes higher up in the hierarchy of namespaces see the real process ID instead of the one for that specific namespace.

2.1.6 User Namespace

The user namespaces perform a similar function to that of the PID namespaces. Inside a user namespace, user and group IDs are mapped from their value inside the namespace to their value outside of that namespace. This means that each container has their own root user (UID 0), but on the host system, this user is not necessarily a root user, it could be any user and might have any other user ID.

2.2 Control Groups

While control groups are implemented similarly to namespaces, they are completely separate from namespaces. Control groups, or CGroups, allow the user to restrict resources in the system. Just like it was the case with namespaces, control groups also divide processes into groups of processes that have the same restrictions as its peers in the group. The control groups available to a Linux 3.18 machine are cpuset, cpuacct, memory, devices, freezer, net_cls, net_prio, blkio, perf_event and hugetlb.

In the context of this document, the interest is mostly toward the hugetlb, devices, and cpuset control groups.

2.2.1 Hugetlb

Using this control group, it is possible to set an upper limit on the amount of hugetlb memory that can be allocated by that specific control group. Any application in this group, trying to allocate more hugetlb memory, will report an out of memory error while there might still be hugepages left on the host system.

As an example, a limit of 4 GB, which equals 4 hugepages of 1G is imposed on the CGroup with name test.

```
# mkdir /sys/fs/cgroup/hugetlb/test
# echo $((4*1024*1024*1024)) > \
    /sys/fs/cgroup/hugetlb/test/hugetlb.1GB.limit_in_bytes
```

2.2.2 Devices

This control group makes it possible to assign specific rights to device nodes in the system. It is possible to assign read, write, or make rights to a device node or a type of a device node. This limits the access to files under the /dev directory.

The below example revokes the default rule that grants access to all device nodes. The second rule then re-enables the device node with the major number 242 and the minor number 2. The processes in the test CGroup can now read (r), write (w), and make (m) this type of device nodes.

```
# echo "a *:* rwm" > /sys/fs/cgroup/devices/test/devices.deny
# echo "c 242:2 rwm" > /sys/fs/cgroup/devices/test/devices.allow
```

This methodology can be used to partition the device resource between control groups.

2.2.3 Cpuset

When reserving specific cores in your system for a specific process, you will most likely also want to enforce this reservation. The cpuset control group specifies the cores that can be used to execute any of the processes in that control group. Other cores will be unusable and when trying to affinitize a process to them, it will fail with an error.

```
# echo “0-7” > /sys/fs/cgroup/cpuset/container1/cpuset.cpus
```

2.3 Container Tools

As containers have evolved on Linux, so has the toolstack used to control the containers. By far the most popular at this time is Docker, which provides an easy to use command line to control the container life cycle. Some of the other tools that can be used to manage containers are LXC/ LXD, Rocket* and SystemD.

In this document, Docker is used since it has an active community and builds container images easily. These container images are a useful feature when launching multiple Virtualized Network Functions (VNFs) performing different tasks on the system without having to change the container image itself.

Docker consists of two components. First, there is the Docker daemon which runs on any system to launch containers on. Then, there is the command line client which talks to the daemon using a REST Application Interface (API).

The REST API exposed by the Docker daemon can be used by other tools as well, and there is great support for Docker in many of the orchestration tools. There even are some software packages purposely built for container management such as Kubernetes* which use Docker on the host systems.

2.4 Issues

Despite being a light weight virtualization alternative compared to hypervisor based virtual machines, these two virtualization methods are not the same just yet. After the general description of how containers work, some of the issues experienced when looking into how to use a container are investigated. Some are more relevant than others to the discussion for usage with NFV but factors such as security cannot be left out from the equation.

2.5 Pseudo Filesystems

Much of the information about kernel structures is read from the /proc and /sys pseudo filesystem. While these are available to containers, they present a problem, these structures represent the host kernel data structures. These data structures in turn represent the state of the entire system and not only the container from which these filesystems are inspected.

Some of the most used userland tools read information from this filesystem instead of making a system call, if there even is a system call. This leads to inconsistent information being used by container VNFs. These operations within the container VNFs can lead to serious security issues although not carried out intentionally. This issue using memory and CPU information available to containers is now discussed.

The amount of free memory and memory usage in general is one of these sets of information that is represented by the /proc filesystem, /proc/meminfo to be more precise. One of the most used utilities in Linux to inspect memory, free, will use the /proc pseudo filesystem to retrieve the amount of free memory. However, since this file represents the state of the entire system and not the container it will display the total amount of memory free to the host kernel. It does this regardless of the limitations imposed on the container it is being called from. This might result in applications reporting that they are out of memory while the free utility still reports the availability of memory.

Trying to retrieve information about the available CPUs to the system through /proc/cpuinfo or /sys/devices/system/cpu will report all the CPUs available to the host kernel, regardless of the cpuset imposed on a container. A process or thread might try to set its affinity to a core it knows is available, but still fail because it is not authorized to access it.

3.0 A Comparison with Kernel-Based Virtual Machine (KVM)

Although the comparison is often made between hypervisor-based virtual machines and containers, they both are very different methods of virtualization. While a hypervisor abstracts hardware and operating system by emulation or pass-through hardware, containers provide operating system virtualization and use resources provided by the Linux host without emulating any of the devices it has attached.

In this section, we will explore the differences between a KVM and a container that is created using the features available in the Linux kernel.

3.1 Choice of Operating System

When running a virtual machine, the expectation is to receive a full emulation of a physical machine from the BIOS to the hardware that the virtual machine uses. Having virtual hardware that resembles normal physical hardware allows installation of any preferred general purpose operating system. This choice of operating system comes at the cost of having to load multiple instances of a kernel into memory (host and guest kernel), but gives great freedom to choose the operating system.

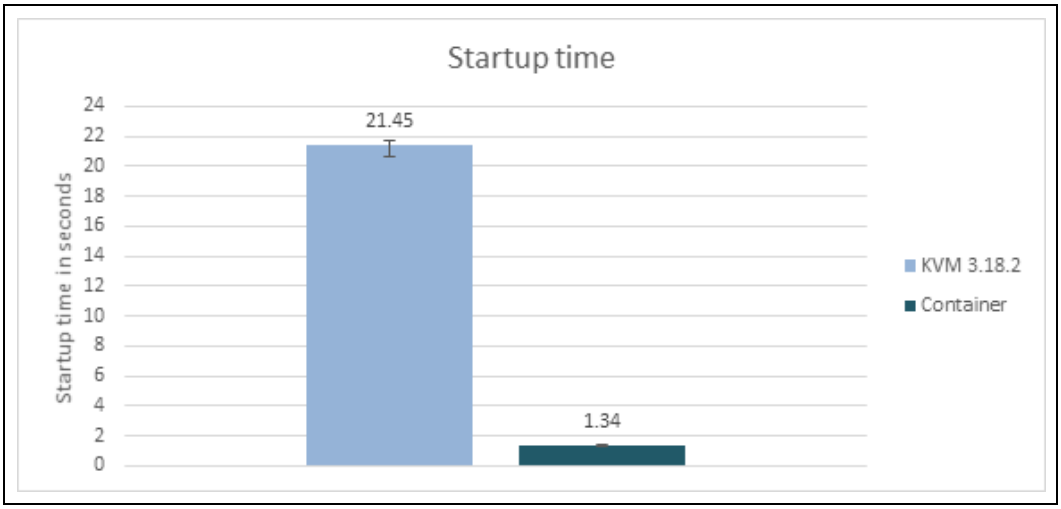
When using containers, the choice of operating system is a bit more limited. Since containers are a form of operating-system-level virtualization, processes are actually isolated instead of virtualizing the physical hardware. Since the isolation of processes happens in the host kernel, and all processes run on the same kernel, there is no duplication of the operating system kernel. The downside of each container using the same kernel is the ability to choose the kernel to use and to run it in the container or load any kernel functionality that was not present on the host system. This means that if a kernel module is to be loaded, it has to be loaded on the host system itself and it will be exposed to every other container running on the same system.

3.2 Startup Time

In this section, the impact of device emulation and full operating system overhead on hypervisor-based virtual machines is shown. Since a virtual machine emulates a full machine, it also goes through the booting of the BIOS and operating system. A container on the other hand does not have to perform any booting ritual, it launches a process in the correct namespaces and it is done.

Since launching a container is essentially launching a process, it is very beneficial for startup time of a container. Comparing the startup times of a default container and a default virtual machine produced the following results on the Intel system using Docker to manage the containers.

Figure 3. Average Startup Time (Seconds) for a KVM Linux Virtual Machine and a Container Over Five Measurements



A script was created that would record the start time of the measurement, then launch the container or virtual machine and wait for a message that will be sent from inside the container or VM. Once it receives this message, it will record the time again and store this as the time at which the container or VM is ready to be used.

When measuring the time before a container would be ready to be used, the script was started that would signal back the process being run. When this script runs, it means that the runtime environment inside the container is ready and the container can be used.

In the VM, the script is attached that would signal back to the host to the INIT script that is launched when the network is up in the VM. At this point, the VM is ready to be used and any service running on it could be launched. This point in time was chosen of the VM startup sequence because this is where most services will be launched that need to be accessible from the outside such as web servers, SSH servers, and so on.

It is important to note here that many optimizations can be done on hypervisor-based virtual machines to reduce the boot time like customizing the kernel, operating system services, device support, and so on. This can drastically reduce the virtual machine boot time.

3.3 Device Emulation

Emulated devices in virtual machines are device models implemented by the hypervisor in the software. This kind of device emulation is not very efficient. Optimizations to device emulation can be implemented using shared memory that optimizes memory operations. The shared memory that is used by DPDK to communicate between virtual machines or from virtual machine to the host is called IVSHMEM, and it is presented to the virtual machine as a PCI device (IVSHMEM Library - DPDK 2.0.0 Documentation, n.d.) (git.qemu.org Git - qemu.git/blob - hw/misc/ivshmem.c, n.d.) (QEMU Patch to support IVSHMEM DPDK driver, n.d.). What happens behind the scene is that Qemu maps the memory that will be shared into its own virtual address space, and then exposes this memory to the virtual machine as a PCI BAR. When writing or reading from this BAR, reading or writing is actually from/to the memory that Qemu mapped earlier.

All of this is possible in virtual machines because there is a hypervisor to set up the device structures in memory required for the guest operating system to detect a device is present. However, containers do not have a hypervisor that is emulating devices, and thus it is not possible to create emulated devices that are not really there. This means that the current technologies like VHost or IVSHMEM will need to be adapted in order to be functional in containers.

3.4 Disk Footprint

When running a virtual machine, provision a disk that the virtual machine can use and then install a full operating system on that disk. Containers can use any filesystem that is mountable as a root directory. If the application you are going to run inside the container does not even require write access to the filesystem, use the host root filesystem in read-only mode. This means that we add absolutely no disk space overhead.

When using Docker, Docker Hub (library: Search Results | Docker Hub Registry - Repositories of Docker Images, n.d.) specifically, there are highly optimized base images available for most of the popular distributions. These images are mostly smaller than 250 MB while a clean install of most operating systems results in multiple GBs of data. On top of these base images, developers can then layer the tools and the applications they want to run. These layers only contain the difference between the current layer and the previous one and thus are usually quite small. If any of the layers are re-used by other containers they are used as a read-only layer and not duplicated on disk. This ensures that multiple containers running the same base image do not fill up the disk space linear to the amount of containers running. Below are the sizes of some of the images available on Docker hub, to compare this with. For example, a minimal CentOS* 7 install which has a disk size of 1.1 GB after a clean install.

Table 2. Size of Docker Images at the Time of Writing

Operating System Flavor	Image Size
Ubuntu* 14.04	188.3 MB
CentOS* 7	229.6 MB
Debian* 7	84.98 MB
Fedora* 21	250.2 MB

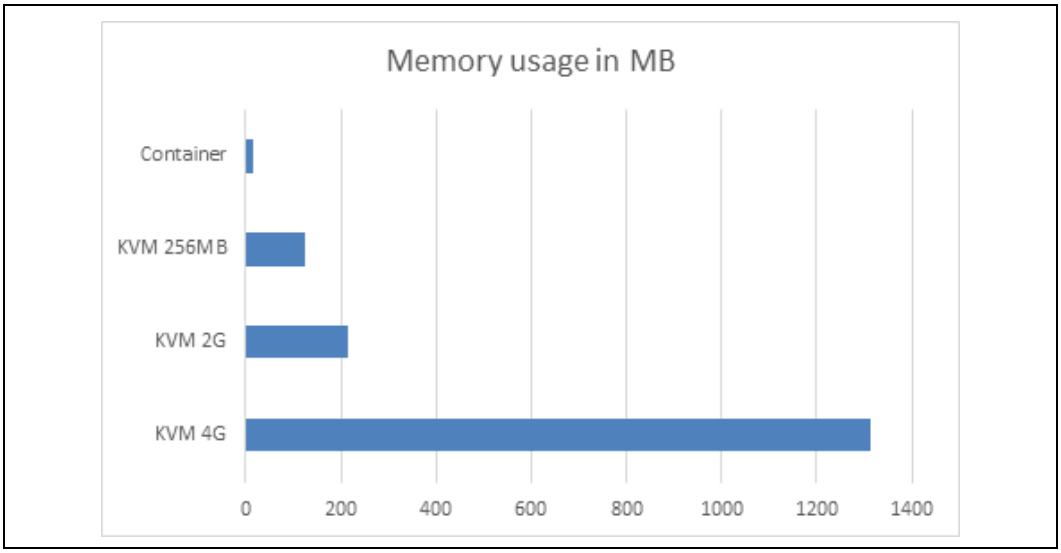
3.5 Memory Footprint

The amount of code that is actually loaded into memory is significantly less with containers compared to virtual machines. This is very easy to explain, only the application is loaded into memory when running containers. When running the sleep command in a container that launched with Docker, 17 MB of memory in total is lost from the memory available on the host system. This number includes the amount of memory necessary to create the containers as well as the memory allocated by Docker to manage the container.

When running a virtual machine, it is not only your application that is in need of memory, the operating system and the hypervisor itself will need memory as well which adds significant overhead to the memory requirements of your application.

For optimal performance inside the Intel virtual machine, Qemu was launched with the --mem-prealloc flag which causes Qemu to allocate all of the memory when the virtual machine is initialized. This means that a virtual machine always uses the total amount of memory allocated to it at minimum. A CentOS 7 virtual machine which has been given 256 MB of memory without pre-allocating it will end up using about 125 MB of memory when booted. Which is over five times more than the equivalent Docker container uses. When assigning more memory to a virtual machine it will start utilizing more despite it being the same virtual machine and sitting idle.

Figure 4. The Memory Used by a Container Versus the Memory Used by a KVM Virtual Machine with Varying Memory Sizes



3.6 Density

Because containers have a smaller footprint, both in memory and on disk, they allow more of them to be launched on a single system as there could live virtual machines on the same hardware. This improves the utilization of the hardware when a machine's CPU is underutilized, but no more guests can be added because of the limitations imposed by the disk or memory that is available.

For VNFs using DPDK, the number of CPUs available in the system is limited, because for optimal performance the same cores are not reused for different VNFs running at the same time on the same system. Because of this, the memory and disk limitations are less of an issue since more memory can be added to modern systems than the addition of cores. As the example below demonstrates, running out of processing power is more likely than running out of memory, even when running multiple VNFs on the same cores.

The maximum amount of memory that can be added to the board used in the Intel tests (ARK | Intel Server Board S2600WTT, n.d.) is 1536 GB while the amount of processors that can be added is limited to two. With the maximum of cores per CPU limited to 18 on an Intel® Xeon® E5-2699v3 CPU (ARK | Intel Xeon Processor E5-2699 v3 (45M Cache, 2.30 GHz), n.d.), this means a total of 36 threads per CPU and 72 threads in the system. Resulting in the possibility of allocating 21 GB of memory per core which is more than plenty for modern applications.

3.7 Configurability

When deploying a DPDK application in a virtual machine, DevOps tools (DevOps tools · GitHub, n.d.) can be used to configure the virtual machine according to a predefined configuration. Or a pre-configured VM image is used. There is no obvious way to pass a configuration to the application that will run in the VM once VM launches. In most of the real world use cases, pre-configured VNFs would be deployed by the orchestrator.

When launching a container, a process is actually being launched, this implies that also to pass a configuration down to the container. Applications that could not flexibly deploy into a virtual machine can now be deployed easily by deploying it in a container. Application configuration can be supplied on the command line when launching a container, or a directory could be mounted containing the configuration of the application inside the container such that the application will transparently pick up the correct configuration.

Being able to deploy VNFs, and applications in general, in such a way allows easier orchestration where the application was not originally created to be orchestrated. Applications can be packaged up as software appliances so that a user should only supply the necessary configuration and not worry about the internals of the appliance.

The benefits of this can easily be demonstrated using an image for a web server, nginx (nginx news, n.d.) (nginx Repository | Docker Hub Registry - Repositories of Docker Images, n.d.), available from the Docker hub. This image deploys multiple web servers without ever having to reconfigure nginx for a different html root directory. Specify the directory to serve when launching the container.

```
# docker run -v /some/content:/usr/share/nginx/html:ro -d nginx
```

Similarly, deploying the Data Plane Performance Demonstrators application (Intel, n.d.) (DPPD) can be done by passing the location of the configuration file for DPPD on the command line. For example, a BNG:

```
# docker run –i –t dppd:v015 http://192.168.0.2/bng.cfg
```

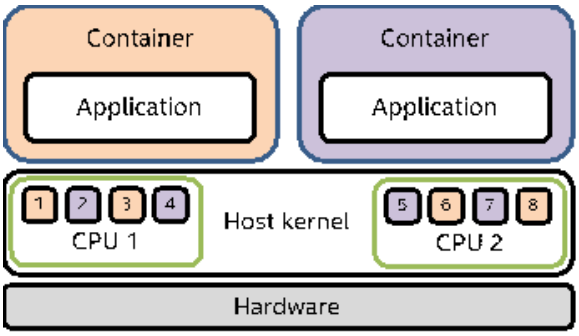
Or a router:

```
# docker run –i –t dppd:v015 http://192.168.0.2/router.cfg
```

3.8 Physical CPU Core Resource Allocation

When discussing the issues in containers a few sections back, the problematic visibility of the host hardware from inside the container was mentioned. While it is an issue from a security perspective, it creates transparency that allows a VNF to be tuned without requiring external information. The most obvious advantage is that inside a container (which has well defined permissions), the layout of the cores that have been assigned to the container are seen in Figure 5.

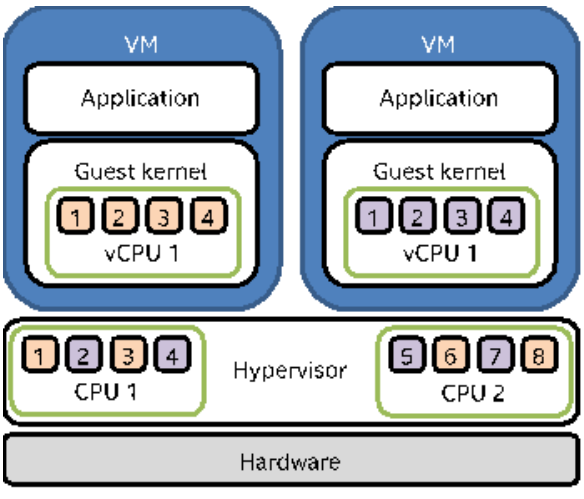
Figure 5. Logical Core Numbers as Visible by Containers



Using traditional virtualization solutions, a virtual machines can only see the virtual CPUs it has been given access to, it cannot determine the physical CPUs that it will end up running on. In order to configure VNFs for optimal performance, careful consideration needs to be given to the hardware resource allocated to the VMs.

For example, if wanting two threads to have efficient cache utilization, allocate and assign them to adjacent cores. But the information of the locality of the physical cores is not available to a virtual machine L2 caches. When using a container, full visibility into the cores received allows tuning the applications better according to the core configuration supplied.

Figure 6. Logical Core Numbers as Visible to Applications Running on the Guest Kernel



3.9 Security

The community has had some concerns with container security since it does not provide the same level of separation between the guest systems and the host. While traditional virtual machines only share the hypervisor and run their own kernels, containers share the entire kernel with each other. The kernel provides for a much bigger attack surface than a hypervisor does. On top of this, a process running as root within the container would also have root privileges outside of the container, until kernel 3.8, should it ever break out. The question has to be raised: how to protect the host from attackers within a container?

The *user namespaces* allow for greater security in containers since kernel 3.8, user IDs can be mapped to a different range of user IDs on the host system. This means that the user or group ID within the container can be different from the ID seen outside of the container.

Specifically, this means that containers can be launched using a regular user instead of root so that when a process would break out of its container, it would not have root privileges on the host system. This reduces the amount of harm a process can do on the host system significantly.

A common belief is that a root account on a Linux machine can do anything. Although this is generally true, it does not have to be. *Capabilities* can be used to specify what a root user can or cannot do. There are in total 32 capabilities available which limit the different actions that a root account can do. An example is *chown*, normally a root user can change the ownership of a file regardless of the permissions set on the file, this can be limited by disabling the *CAP_CHOWN* capability. Another example is the capability to create device nodes on the system, this can be controlled using *CAP_MKNOD* (capabilities(7): overview of capabilities - Linux man page, n.d.).

Libseccomp allows the drop of system calls from the syscall table that should not be called by a process. For 64-bit containers, there are in total 649 system calls in Linux, of which 338 (Linux Syscall Reference, n.d.) are 32-bit system calls which can safely be disabled on a pure 64-bit system (Chapman, n.d.).

Limiting the system calls that can be made from a container seriously limits the attack surface of the kernel. After all, these system calls are how processes interact with the kernel and are able to exploit potential bugs in the kernel.

An extra layer of security is integrating *SELinux*, or the similar *APPArmor*, to contain processes within the filesystem that has been provided. Implementing SELinux security ensures that even when a process breaks out of its container it will be limited in the damage it can cause. SELinux provides mandatory access control, meaning that access to a resource should be explicitly granted to a *subject*, which can be a process or thread. It allows for precise control over the permissions granted to processes than the default read-write-execute model that Linux provides.

In the context of containers, SELinux can be used to label the filesystems that a container will be using and grants processes in the container only access to resources that have this label. Should a process now break free of the isolation, it will have to exploit SELinux before being able to infect other containers or the host operating system (https://www.youtube.com/watch?v=zWGFqMuEHdw, n.d.).

3.10 Summary

Table 3 provides an overview of all the topics discussed above to provide an easy comparison of hypervisor-based virtual machines and containers depending on the requirements. For performance data, refer to the section that has the corresponding name. A performance comparison between the two can be found at the end of this document, where some VNFs were implemented on the host, in a virtual machine, and in a container.

Table 3. Comparison Between Docker Containers and KVM

	Containers	Hypervisor-Based Virtual Machines
Choice of Operating System	Variant of host OS only	Any
Startup Time (Unoptimized Default)	1.5s	21s
Hardware Abstraction and Device Emulation	No	Yes
Disk Footprint	Small	Large
Memory Footprint	Small	Large
Density	High	Low
Configurability of Application	Flexible	Complex
Tunability from Guest	Insight into host	Black box machine
Security	Not mature and complex	Mature Security models

Note: Table 3 only covers the comparison between containers and virtual machines discussed in the paper. Topics like orchestration, management, application design, and so on are out of scope of this document.

4.0 Optimizing Performance Using a dyntick Linux Kernel

When running a VNF on a virtual machine, the configuration of the virtual machine will be optimized in order to optimize the performance gained from the virtual machine. For example, the Qemu threads will be pinned to a CPU to prevent them from being migrated by the scheduler to a different core. The guest's memory will be backed by hugepages to prevent TLB misses when the virtual machine tries to access large amounts of memory.

As containers run on the operating system, Intel investigated if the performance could be increased of the containers by optimizing the kernel. As only one application thread per core in the system will be running, the full dyntick feature of the kernel seemed like a candidate for improvement. This combined with a slower timer tick (100 Hz) could increase the amount of CPU time an application receives because it would not be interrupted by the kernel as much.

The full dyntick feature allows a task to monopolize a core when it is the only task running on that core. When this occurs, the kernel disables the timer that would invoke the scheduler and thus allow a task to keep executing without being interrupted.

This feature is not enabled by default on CentOS 7 so the kernel was recompiled with support for it by enabling the CONFIG_NO_HZ_FULL=y flag and friends. The option can be enabled using make menuconfig as well and lives under General setup, Timer subsystem.

5.0 Setup of a DPDK-Enabled Container

Running DPDK applications in a container requires some configuration before actually starting running applications in containers. Here the configuration that is required is discussed and why it is required.

5.1 Configuring the Host System

5.1.1 Loading the igb_uio Kernel Module

DPDK uses the igb_uio kernel module driver to communicate with the NIC mainly for configuration; however, since this driver is not part of the Linux kernel it has to be loaded by the host. Only the host has the capabilities to load a kernel module since it would be a security concern to let any container load a kernel module. The host and all the containers on the system share the same kernel.

Because of this limitation, load the igb_uio driver on the host and bind the network interface cards to the driver on the host as well. Any further communication with the NIC can now happen using the /dev/uioX device node that corresponds with the NIC used.

5.1.2 Loading the vfio Kernel Module

Since the release of DPDK 1.7, it is possible to use the VFIO instead of using the IGB_UIO driver (Intel, 2014). While IGB_UIO is not available in the upstream kernel, VFIO is. This opens the possibility to using DPDK applications without the need to load application-specific-kernel modules on the host.

Binding the VFIO driver to the network adapters available in the system still requires access to the host system. Giving a container permission to bind adapters to a driver would require full privilege to the specific driver, and a container would be able to bind or unbind any device to this driver.

5.1.3 Loading the KNI Kernel Module

The KNI kernel module (Kernel NIC Interface - DPDK 2.0.0 documentation, n.d.) or kernel NIC interfaces, allow a program to create a network interface that is accessible through standard Linux network APIs but that is backed by DPDK rather than by the Linux networking core. If an application would like to use this kernel module, it would have to be loaded by the host as well, but that is not the only problem for the KNI kernel module. When creating a KNI adapter from inside the container, that adapter will show up in the host namespace rather than that of the container. The container will need assistance from the host in order to access the Linux interface that has been created by the kernel. The host will have to find out to which container the interface belongs and then assign it to that container.

5.2 Creating a Docker Image

Since the philosophy behind Docker is to create images and not log in to each container to configure it, a set of images here will be created. The images described are base images that allow building DPDK applications easily on. Since running inside a container also requires some patches to DPDK, at this stage they will be included.

To create a Docker image, create a Dockerfile that describes the steps required to build the end result. The Dockerfile contains commands such as RUN or ADD which each create a new layer in the image hierarchy. An example skeleton Dockerfile that can be used to generate a DPDK base image is detailed below.

1. Start from a CentOS 7 image and use the /root directory for the application.

```
FROM centos
WORKDIR /root
```

2. Install make, gcc, patch and numactl.

```
RUN yum install -y tar make gcc patch numactl && yum clean all
```

3. Set up some environment variables required by DPDK and by ourselves.

```
ENV RTE_SDK=/root/dpdk
ENV RTE_TARGET=x86_64-native-linuxapp-gcc
ENV RTE_VERSION="1.7.1"
```

4. Download DPDK, disable the IGB_UIO driver and KNI driver since these are kernel modules which would have to be loaded into the kernel, and this is not possible from inside a container.

```
RUN curl http://dpdk.org/browse/dpdk/snapshot/dpdk-${RTE_VERSION}.tar.gz | tar -xz
&& \
    mv dpdk* dpdk && \
    rm -Rf dpdk/app && \
    sed -i 's/ROOTDIRS-y := scripts lib app/ROOTDIRS-y := lib/' dpdk/GNUmakefile &&
\
    sed -i 's/CONFIG_RTE_EAL_IGB_UIO=y/CONFIG_RTE_EAL_IGB_UIO=n/' dpdk/config/
common_linuxapp && \
    sed -i 's/CONFIG_RTE_LIBRTE_KNI=y/CONFIG_RTE_LIBRTE_KNI=n/' dpdk/config/
common_linuxapp
```

5. Apply the hugetlbfs patch to detect the allowed size correctly.

```
ADD hugetlbfs_statvfs.patch /root/dpdk/hugetlbfs_statvfs.patch
RUN cd dpdk && patch -p1 < hugetlbfs_statvfs.patch
```

Whenever this image is used for a DPDK application, build the DPDK source.

```
ONBUILD RUN cd dpdk && make config T=$RTE_TARGET && make -j && make -j install
T=$RTE_TARGET
```

5.2.1 Detecting hugetlbfs Mount Size

The hugetlbfs patch enabled DPDK running in the container detect the size of the hugetlb mountpoint rather than the total amount of hugepages in the system. Whenever a DPDK application launches, it detects the amount of free hugepages, then it will try to allocate all of them so that it can select the hugepages that are mapped on the desired CPU socket. However, to limit the amount of memory that a single container can allocate, set a maximum to the amount of memory that is allocable from the hugetlbfs. Configure this limit using both the hugetlb CGgroup and using the size option on the mountpoint. This last option can easily be detected using the statvfs function, and that is exactly what this patch implements.

5.3 Launching the Container

Intel uses Docker to launch and manage the containers; however, the default setup used by Docker is not compatible with DPDK applications. The PCI devices are masked out so that the applications in the container does not have access to them. A volume is provided on which hugepages can be created and finally masking out the CPUs that are not in the cpuset of that specific container.

5.3.1 Masking PCI Devices

Similar to the situation described earlier, PCI devices are exposed to applications running inside the container even if they do not have access to them. This again causes the DPDK EAL to think that the PCI device is present and usable. However, this is not the case, so the devices are hidden to which the container should not have access. To overcome this problem, the PCI devices on /sys can be masked. The /sys/bus/pci/devices directory contains only symbolic links to the actual devices structures, the entire directory can be masked with an empty directory and then populate this new empty directory only with the symbolic links for the devices that are accessible from inside the container.

Whenever DPDK scans this directory looking for available PCI devices, it will only find the devices for which the necessary symbolic links were provided and thus will not access any device to which it has not been granted access.

When it comes to assigning network interface cards to the container, there is another problem, knowing which device node in the /dev folder belongs to which PCI device. This information can be retrieved from the device information under /sys/bus/pci/devices/<pci-dev>/uio once the device has been bound to the igb_uio device driver that is used by DPDK.

5.3.2 Masking CPU Cores

The Linux kernel exposes the available CPUs and their feature through sysfs which is mounted on /sys when running a Linux system. On this filesystem, find a folder for each logical CPU core available on the system under the directory /sys/devices/system/cpu/. The DPDK environment abstraction layer uses this directory to detect the logical cores that can be used, specifically it tries to access the /sys/devices/system/cpu/cpuN/topology/core_id file to see if core N is available. It will try this for each core where N is between 0 and 127.

In order to prevent DPDK from detecting and trying to access cores that are not in the container cpuset, an empty directory will mounted over the cpuN directories of the cores that should not be available to the application in the container. This will ensure that the DPDK is unable to read any file that exposes information about this logical core, including /sys/devices/system/cpu/cpuN/topology/core_id.

This change to the container will not change the behavior of the kernel in any way, but might cause incorrect behavior for other applications that would run using this setup. The Data Plane Performance Demonstrator application performs a similar check to that of DPDK to find out which

physical core a logical core belongs, but fails to check if the file topology/core_id even exists. In such a case, the application has to be patched itself to perform a check whether the file exists or else it will crash when attempting to read from that file.

5.3.3 Mounting a hugepage Volume

When running a virtual machine it is possible to specify the maximum amount of memory an operating system instance can use. The same is possible for a container, but when using a container that does not include the memory provided by the hugetlbfs. There is a way of specifying the maximum amount of hugepages that a container can retrieve.

In order to achieve this, a mount point is created that is unique for each container and mounts a new hugetlbfs on each with the size option set to the maximum amount of memory that can be allocated from the hugetlbfs.

Mounting hugetlbfs with 1 GB pages and a maximum of 4 GB would look something like the following:

```
# HUGEPAGE_1G=$((1024*1024*1024))
# HUGEPAGE_MEM=$((4*$HUGEPAGE_1G))
# mount -t hugetlbfs -o pagesize=$HUGEPAGE_1G,size=$HUGEPAGE_MEM \
    none /mnt/container1/hugedir
```

Attaching this mount point to the container will effectively limit the amount of hugepages that each container can allocate. On top of that, this approach ensures that containers on a single system cannot see, read, or write any of the hugepages of another container. And when this is desirable, the hugedir can be attached of one container to another without endangering any of the other containers on the system.

For additional security, the HugeTLB CGroup settings will also be set such that the memory limit will be enforced when a mmap system call is performed. Enabling this CGroup setting is as simple as creating a new group in the hierarchy, assigning the process ID to the task and then setting the limit.

Assigning a container to the container1 group and setting a limit of 4 GB looks like this:

```
# CGROUP=/sys/fs/cgroup/hugetlb/system.slice/container1
# mkdir -p $CGROUP
# echo pid-of-container > $CGROUP/tasks
# echo $((4*1024*1024*1024)) > $CGROUP/hugetlb.1GB.limit_in_bytes
```

When all this configuration has been done, DPDK needs a patch to read the size of the filesystem the hugetlbfs is on. Otherwise, it will try to allocate all of the free hugepages in the system. It is more than likely that the amount granted to this container is smaller than the amount of free hugepages available on the host system.

After the limits have been configured, a last command is needed to ensure that the memory received is allocated on the correct NUMA node of the system. DPDK will refuse to launch if it cannot find enough memory on the socket that it wants to use. It is up to the kernel to choose the NUMA node to allocate memory from, but since limiting the amount of memory a container can retrieve, only pages from the wrong NUMA node may be present.

To signal the kernel that memory requested by the application should be allocated on a specific NUMA node, *numactl* can be used to indicate the preferred NUMA node for the application. When launching the DPDK application inside the container, this will run:

```
# numactl –p0 testpmd –c 0xf –n 4
```

However, *numactl* is not a package that is included with the docker images of CentOS and setting the NUMA node for the container from the host needs to be done. This can be done by using the *cpuset* CGroup. Setting the *cpuset.mems* will limit the application to memory located on the nodes

specified in *cpuset.mems*. Limiting the memory of a container to NUMA node 1 is accomplished with the following command, memory that potentially already resides on NUMA node 0 will be migrated to the new node.

```
# CGROUP=/sys/fs/cgroup/cpuset/system.slice/container1
# echo 1 > $CGROUP/cpuset.mems
```

5.4 Inter-Virtual Machine Shared Memory

When there is a need for high performance communication between virtual machines, shared memory can be used. Shared memory allows the sharing of packet data and other DPDK structures without copying them between virtual machines.

When using KVM, the memory is shared through a PCI device that Qemu generates. This device exposes a PCI BAR that represents the memory on the host. The PCI BAR of these virtual devices in different virtual machines map to the same memory on the host machine and thus expose the same section of memory.

Containers are unable to use the same approach to expose the memory regions and due to the use of namespaces they are also unable to communicate the different memory regions with each other like being able to do on the host.

To support these use cases, a patch to DPDK is needed that uses the same memory with the least amount of configuration. That is why the option to DPDK was added, to read shared memory directly from the hugepages on the hugetlbfs. Using this patch, a single hugetlbfs mountpoint between different containers can be shared and uses the pages on this mountpoint to access the shared memory structures.

Instead of passing the Qemu options to configure the shared memory to Qemu, it is now passed on the command line of the DPDK application and the IVSHMEM structures will be loaded by DPDK similarly to the approach used when running in a virtual machine.

6.0 Proof of Concepts

The goal of these proof of concepts is to illustrate steps involved in porting example virtualized network functions in to a containers environment without any loss of functionality or performance. The hardware used for these proof of concepts consists out of two machines, one is used for packet generation and the other is used as the system under test. The System Under Test (SUT) is the machine that will be running the network virtualized functions while the other machine will be used to generate the packets that will be processed by the system under test. The SUT consists of a dual socket mainboard with two Intel® Xeon® E5-2690v3 processors with 32 GB of memory each, which results in a total of 64 GB of system memory. The system is equipped with two Intel® 82599ES dual port 10 Gb Ethernet controllers. On this system, 50 hugepages will be requested of 1 GB in size to be used with Qemu and DPDK applications. The machine used to generate the packets is a dual socket machine with two Intel® Xeon® E5-2690 processors each with 16 GB of memory. The system is equipped with the same amount of dual port 10Gb Ethernet Intel® 82599ES controllers. Of the total of 32 GB of memory, 24, 1 GB hugepages will be requested to be used by the packet generator. In order to exercise the most amount of control over the CPU cores for the proof of concepts used, the isolcpus=kernel option is added to the command line such that no core on socket 0 will be used to schedule user tasks on.

For these tests, we will be using the Intel® Data plane Performance Demonstrators v015 (Intel, n.d.) on the system under test to perform various network functions. DPPD has many different components which can be combined into a VNF for a specific use case.

On the generation side, Pktgen (Wiles, n.d.) will be used to generate the traffic that being sent to the system under test. All of the scripts used with Pktgen are provided with the DPPD source code.

Figure 7. Test Configuration with Containers

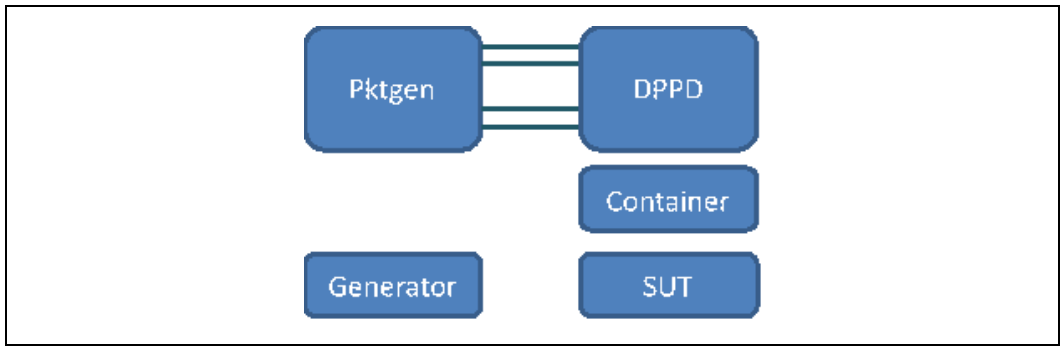
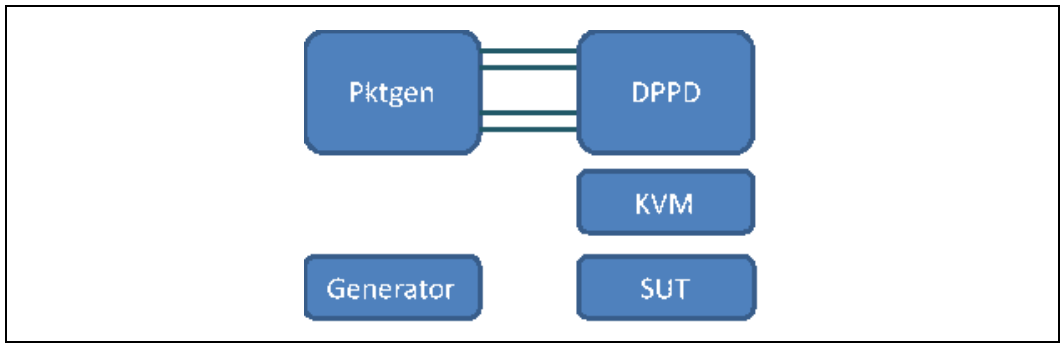


Figure 8. Test Configuration with KVM



6.1 Example DPDK Layer 2 Forwarding

For this test, two network interfaces will be used to receive packets of 64 bytes in size and two different network interfaces to send them back out through. The entire task of forwarding these packets is assigned to a single core of the CPU because one core would be limited by the network interface rather than by processing power or memory bandwidth.

6.1.1 Without Modification

One of the tests, none, named after the mode in DPPD, forwards a packet from one interface onto the next without ever touching the packet. This test essentially only moves pointers around in system memory and the results should not differ much between virtualization methods.

6.1.2 Updating Ethernet Addresses

The other test, l2fwd, will perform almost the same functionality as the “none” test except that it will update the source and destination mac address inside the packet before transmitting it on a different network interface. This causes the application to read and write to the packet and will require packets to be moved into higher level caches which might become a bottleneck.

6.2 Example DPDK Layer 3 Forwarding

Layer 3 forwarding uses the IP information inside the packet to determine the destination network interface to send the packet out through. This requires a lookup of the IP address in a table that resides in memory and thus adds extra overhead on top of the necessary updates to the packet itself.

6.3 Example DPDK BNG Application

The last test case is that of the broadband network gateway, which moves traffic coming from the CPE onto the core network. Data coming from the CPE is encapsulated using an 802.1ad (QinQ) Ethernet header while the traffic on the core network is using GRE traffic tagged with an MPLS tag.

Figure 9. Packet Layout for Traffic Coming from the CPE

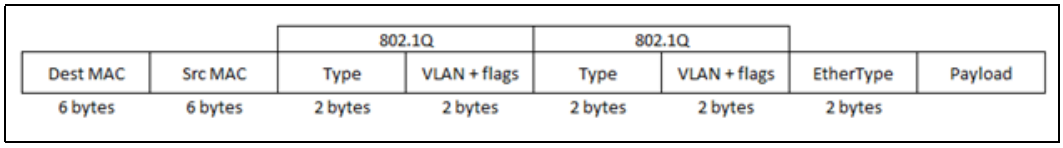
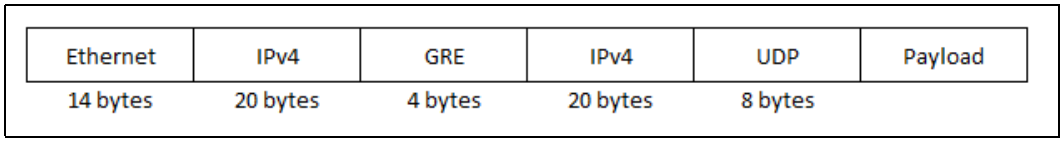


Figure 10. Packet Layout for Traffic Coming from the Core Network



The BNG VNF consists of tasks performing MPLS untagging, load balancing, and QinQ encapsulation based on the GRE header for traffic coming from the broadband network. For the traffic coming from the CPE, there are tasks that perform load balancing, QinQ decapsulation, and routing. The QinQ decapsulation and QinQ encapsulation tasks are run on only two cores in order to create a bottleneck in the system that will perform differently when run inside a virtual machine, container, or on the host.

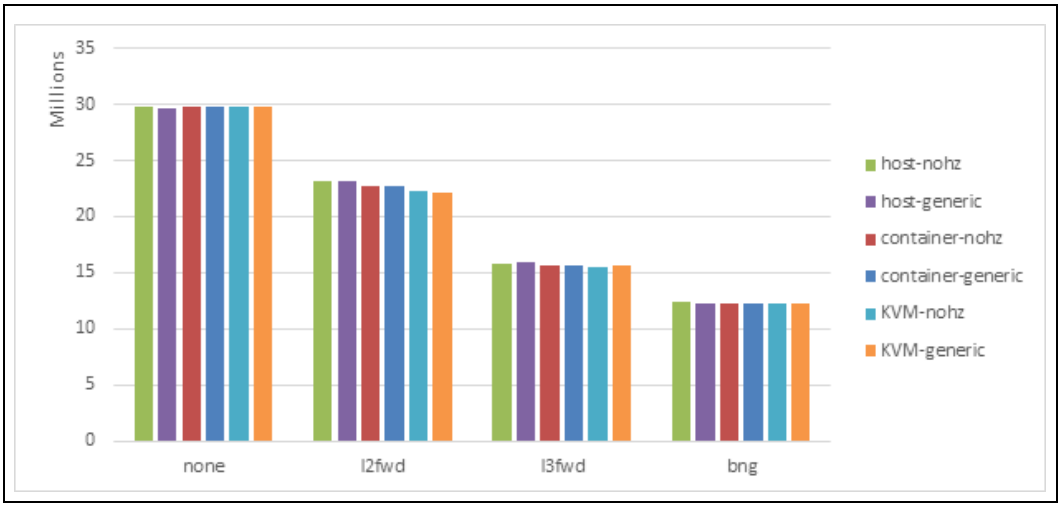
6.4 Results

6.4.1 Throughput

As the graph in Figure 11 suggests there seems to be little performance gain from the use of containers versus virtualization using KVM in most cases. In all the tests, the resulting packets per second that the system is able to push through the system is stable and similar. The only test in which to see a degradation of performance for both containers and KVM was the l2fwd test where containers would do worse than the host system and KVM would do worse than the containers.

Intel looked into the usage of a full dyntick kernel versus a normal one in each of these cases and can conclude from the tests performed that there is no advantage for the VNF use cases tested for.

Figure 11. Packets per Second That a VNF Can Process in Different Environments



6.4.2 Latency

When looking at the latency introduced during the "none" task or the simple forwarding, the KVM introduces slightly less latency on average. However, when looking at the maximum latency introduced, the KVM virtual machine latency jumps up to 457 microseconds which means that there is more jitter when using these VNFs. This difference in maximum latency can be attributed to the additional time spent in the hypervisor when an interrupt is fired. Since timekeeping in Linux is done through the hardware timers, an interrupt is fired rather regularly, and thus can also impact the latency introduced on packets passing through at the time of the interrupt. VM-exit and VM-resume operations are still costly on current generation CPUs and they introduce latency as well.

Table 4. Latency Introduced by Moving Packets from One NIC onto Another

Environment	Average Latency (µs)	Maximum Latency (µs)
Host	11.4375	53
Container	11.3955	56
KVM	11.1735	457

When looking at the latency introduced during the "l2fwd" task, containers actually introduce a higher latency on average than KVM or the host does. However, the virtual machine is less stable and has a much higher maximum latency. The added latency can be attributed to kernel code that is not fired when running in the root control group. For some of the control group functionality, the Linux kernel branches where it would not when running on the host (kernel/git/torvalds/linux.git - Linux kernel source tree, n.d.). This causes more code to be executed and could also confuse the branch prediction logic of the processor.

Table 5. Latency Introduced by Layer 2 Forwarding

Environment	Average Latency (µs)	Maximum Latency (µs)
Host	493.148	565.441
Container	530.065	599.067
KVM	518.032	974.435

More optimization to reduce the CPU interference because of host and guest kernel threads needs to be done in order to reduce the maximum latency spike, see in the KVM (virtual machine) case.

7.0 Summary

Virtualization technology is the key ingredient that is pushing the rapid adoption of network function virtualization. Advancements in operating system-based virtualization (containers) and hardware abstractions-based virtualization (hypervisor based virtual machine) will provide NFV system designers options to choose the type of virtualization that is best suited for the workload.

With the backing of a rich ecosystem, container-based virtualization is going to get better in addressing some of the security, deployment, and management issues. Containers will also get the benefit from utilizing existing hardware virtualization assists like Intel® Virtualization Technology (Intel® VT) for Directed I/O (Intel® VT-d).

This paper details some of the platform and infrastructure level design considerations and contrasts it between container and virtual machine-based VNFs. The performance benchmarks presented here are the out of the box performance data and with more scope for optimization.

§ §